HIOS: A Host Interface I/O Scheduler for Solid State Disks

Myoungsoo Jung[1], Wonil Choi[1], Shekhar Srikantaiah[2]*, Joonhyuk Yoo[3], Mahmut T. Kandemir[4]
[1]Department of EE, The University of Texas at Dallas,
[2]Qualcomm, [3]College of ICE, Daegu University, [4]Department of CSE, The Pennsylvania State University
{jung, wonil.choi}@utdallas.edu, ssrikant@qti.qualcomm.com, joonhyuk@daegu.ac.kr, kandemir@cse.psu.edu

Abstract

Garbage collection (GC) and resource contention on I/O buses (channels) are among the critical bottlenecks in Solid State Disks (SSDs) that cannot be easily hidden. Most existing I/O scheduling algorithms in the host interface logic (HIL) of state-of-the-art SSDs are oblivious to such low-level performance bottlenecks in SSDs. As a result, SSDs may violate quality of service (QoS) requirements by not being able to meet the deadlines of I/O requests. In this paper, we propose a novel host interface I/O scheduler that is both GC-aware and QoS-aware. The proposed scheduler redistributes the GC overheads across non-critical I/O requests and reduces channel resource contention. Our experiments with workloads from various application domains reveal that the proposed scheduler reduces the standard deviation for latency over state-of-the-art I/O schedulers used in the HIL by 52.5%, and the worst-case latency by 86.6%. In addition, for I/O requests with sizes smaller than a superpage, our proposed scheduler avoids channel resource conflicts and reduces latency by 29.2% compared to the state-of-the-art.

1. Introduction

NAND flash-based SSDs are being employed as both replacements to spinning disks and an intermediate layer in the storage hierarchy, owing to their superior performance and lower power consumption as compared to disks [8]. Consequently, a wide range of execution platforms, from embedded systems to high-performance and enterprise computing systems, employ SSDs as an intermediate layer in the form of a storage cache or a write log, or as a substitute for legacy disks [9, 10]. An important concern when SSDs are used as an intermediate layer is that their performance can be influenced by various complexities posed by the Flash Translation Layer (FTL), including data-mapping, garbage collection (GC), and wear leveling [14, 16, 41, 28]. As a result, SSDs can be plagued by enormous *performance variations* across requests, depending on whether these underlying complexities could be appropriately managed or not [21, 13]. A host-level buffer cache or I/O scheduler may help to mitigate such variations by changing access patterns and/or buffering requests. However, such approaches experience difficulty in hiding storage-level GC latencies. Especially, a GC-oblivious write cache can suffer significantly from high device latencies in the I/O path [21, 20].

Consider the comparison, shown in Figure 1a, between the latencies of seven commercial SSDs (PCI-E, SATA3, and SATA2 based) and four different commercial disks (SATA2). As can be seen from the figure, although the SSDs have significantly lower *average latencies* as compared to the disks, their *worst-case latencies* can be much higher than disks. An important contributor to this is the GC, which is one of the critical bottlenecks in the FTL that cannot be easily hidden [11, 27, 42]. Channel conflicts are another reason for the high worst-case latencies in SSDs. Figure 1b plots the breakdown of the total I/O time with a first-in first-out (FIFO) scheduler implemented in a new SSD. One can see from these results that the overheads that arise from GCs and resource contentions (caused by channel resource conflicts) contribute, respectively, to 27.1% and 8.7% of the total I/O time. An important reason for this high contribution from GCs and resource contentions is that the existing I/O schedulers [29, 24] are optimized to reduce the impact of random accesses in an context of disks and are not aware of the internal details of an SSD architecture. When the same I/O schedulers are adopted in the host interface logic (HIL) in an SSD (as is commonly done in commercial SSDs today), they are oblivious to the performance overheads that are specific to SSDs, such as GC overheads and channel resource contention, which can in turn violate quality of service (QoS) requirements.

Observing that the HIL is an ideal location in an SSD device to bound these extra GC and resource contention related overheads, we propose a novel *host interface I/O scheduler*,

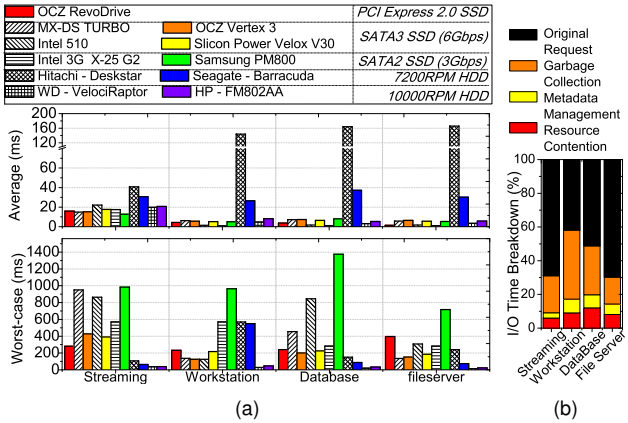
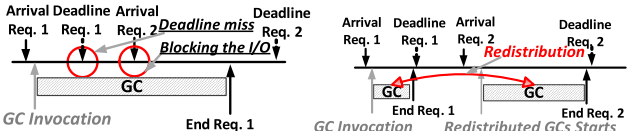


Figure 1: (a) SSD vs. disk comparison using four types of workloads generated by Intel Iometer [18] and (b) I/O time breakdown. "Metadata Management" are overheads to maintain data consistency and reliability, and the portion marked as "Original Request" represents the true NAND access latency for the I/O requests.

*His work is completed when he was with Penn State.

978-1-4799-4394-4/14/$31.00 © 2014 IEEE



(a) a conventional I/O scheduler. (b) our proposed HIOS.

Figure 2: Behavior comparison between a conventional I/O scheduler (a) and our proposed HIOS (b). Since the conventional I/O scheduler is oblivious to GCs under the Request 1, it violates QoS in terms of meeting the deadline. In comparison, HIOS redistributes the overheads over other requests with positive slack (in this case Request 2) and enables all requests to meet the deadline.

called *HIOS*, that is both GC-aware and QoS-aware. The main **goal** of this HIL-based scheduler is to ensure that the overheads associated with each I/O request are bounded within a deadline. The key insight behind the design of HIOS is that a QoS-aware and GC-aware scheduler can schedule requests that are "time critical" when the host-interface on the SSD knows that the SSD is free from GC execution. To achieve this, our scheduler determines the *slack* available to the non-critical requests and schedules the GC related activities such that the GC overheads are distributed over these non-critical requests, instead of critical requests. This paper makes the following **contributions**:

• We propose HIOS, a novel QoS-aware scheduler for SSDs. HIOS can predict critical I/O operations that cannot meet a device-specific deadline (specified at the configuration time), called *negative activities*, and distribute their overheads across non-critical I/O operations. In contrast to existing I/O schedulers, HIOS is aware of specific characteristics of the underlying SSD device like channel resource conflicts and GC overheads that create negative activities. The impact of GC on deadlines is illustrated in Figure 2a for the case of a conventional I/O scheduler. In HIOS, the overheads of negative activities are redistributed by *stealing slack-times* from non-critical I/O operations, as shown in Figure 2b. HIOS significantly reduces the variations across the latencies of different requests in addition to satisfying the deadline requirements of the requests (i.e., the worst-case latencies do not exceed the specified deadline). The overheads of resource conflicts are further reduced by employing an I/O reordering scheme based on an *Empty Channel First* (ECF) principle. To the best of our knowledge, *HIOS is the first device-level host interface I/O scheduler designed exclusively to exploit characteristics specific to SSDs*.

• We present a comprehensive experimental evaluation of HIOS with a range of applications, and compare it against four state-of-the-art I/O schedulers. The results collected reveal that HIOS reduces the standard deviation of latency by 52.5%, and the worst-case latencies by 86.6%, over the state-of-the-art I/O schedulers used in commercial systems. In addition, for I/O requests with sizes smaller than a superpage,[1] our proposed scheduler avoids channel resource conflicts and reduces write latency by 17.5% in the case of random accesses and 29.2% in the case of sequential accesses, compared to a FIFO-style I/O scheduler in the HIL.

[1] A set of multiple physical pages from several NAND chips across channels combines to form a large logical page called "superpage" [10, 8, 41].

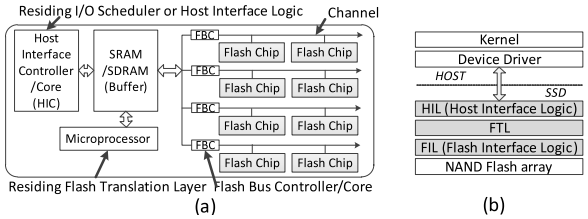


Figure 3: Internal block diagram of a SSD (a) and software stack in SSD (b). An I/O scheduler in the HIL can work concurrently with other software modules.

2. Background

2.1. SSD Organization

As shown in Figure 3a, an SSD is generally composed of a number of flash chips, multiple I/O buses (called *channels*), and three different types of controllers/cores. A flash chip is composed of multiple *pages*, which is the smallest unit to read/write. A page write requires an *erase* operation to a set of pages, collectively called a *block*, which is the smallest unit for an erase operation. The software stack of an SSD is depicted in Figure 3b: 1) a microprocessor is responsible for executing a *flash translation layer* (FTL), which is a software module hiding the idiosyncrasies of the underlying flash. 2) a host interface controller/core executes *host interface logic* (HIL), a software component atop of the FTL providing interface-level compatibility with disks [28], and 3) flash bus controllers/cores handle *flash interface logic* (FIL), which is a software module beneath the FTL issuing flash memory transactions to underlying flash chips associated with one or more channels. Modern SSDs take advantage of I/O parallelism by exploiting multiple channels and cores (e.g., MCX-830 employs 3 cores).

2.2. Internal Parallelism

In order to improve the I/O performance, an SSD can exploit three types of parallelism: 1) *interleaving*, 2) *striping* and 3) *pipelining*. Channel interleaving handles several I/O requests in parallel by using independent flash bus controllers. In Figure 4, for example, two I/O requests in a queue can be simultaneously issued to two different channels. In channel striping, a request can be spread across different flash bus controllers, as illustrated in Figure 4b. These *sub-requests* are then executed by these flash bus controllers in parallel. Lastly, since a request is composed of several NAND flash commands, an SSD can interleave requests by exploiting pipeline parallelism across many NAND flash chips connected to a channel, as shown in Figure 4c. In general, a combination of these three strategies [6, 15] is used in FTL or FIL to fully utilize channel resources and maximize system performance, as shown in Figure 4d. Even though in principle these strategies can improve performance, in practice one can observe a very poor performance when the requests in the I/O queue experience *channel resource contention*. This is because state-of-the-art

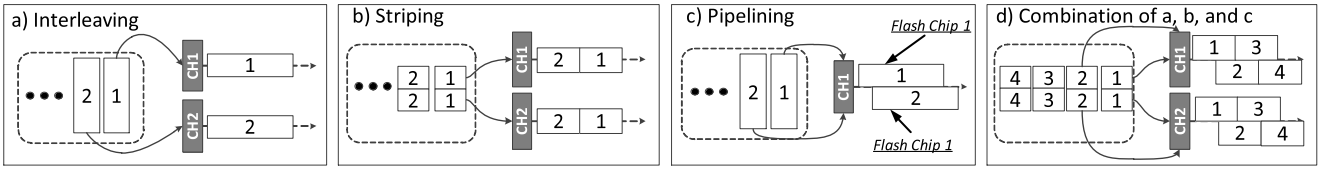


Figure 4: Three major types of parallelism within an SSD. Since each Flash Bus Controller (FBC) handles a channel, these different types of parallelism lead, in principle, to high-performance. However, if a channel is already allocated to other requests, issuing further requests to it generates a serialized access pattern that can result in performance degradation.

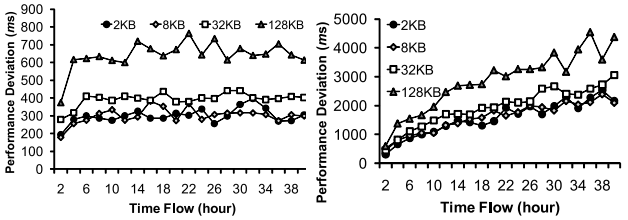
I/O schedulers are not aware of SSD internals. In contrast, our proposed HIOS manages channel resource contention by being aware of the internal hardware configuration and by interacting with the flash bus controllers. Specifically, it changes the order of the I/O requests in a device-level queue and serves first the I/O request heading to an available channel, instead of ones that target a busy channel. This in turn enables the underlying software modules (FTL/FIL) serve I/O requests with minimal resource contention, alleviating a major performance bottleneck.

2.3. Firmware and Garbage Collection

Host Interface Layer. Existing interfaces such as SATA [35] and SAS [38] and associated device-level queues like NCQ [19] and TCQ [38] allow an SSD to schedule I/O requests with no host-level software intervention. To achieve this, a small size *tag*, which is basically an identifier for I/O requests, is sent to the SSD before the actual I/O request service and corresponding data transfer begin. In these thin interface protocols, HIL is responsible for parsing I/O commands, hand-shaking, buffer and data movement handling. Since the raw communication protocol and submission of such parsed I/O commands to underlying layers are managed by HIL, it is an *ideal location* to make device-specific scheduling decisions for I/O requests. In contrast to HIL, host-side modules are oblivious to SSD specific characteristics, and the underlying software in an SSD have no knowledge of the interface protocols. In current practice, a FIFO-style scheduler is employed in HIL with no preference (e.g., no QoS concern) [28].

Flash Translation Layer. FTL mainly handles two limitations of flash: 1) *erase-before-write* and 2) *in-order update*. First, no write request is allowed to flash locations where data have already been written; the target location needs to be erased first. To address this erase-before-write characteristic, FTL has to prepare in advance empty block(s) that have been erased, and forward incoming write I/O requests to such blocks rather than writing them to the original destination blocks. Second, modern flash does not allow out-of-order updates within a block [34, 33]; instead, an FTL writes data in-order using another the empty block [25]. FTL remaps the addresses between logical and physical for both the empty block allocation and out-of-order update writes. Since securing erase blocks is a key factor as far as FTL performance is concerned, modern SSDs typically reserve 3% $\sim$ 8% of total storage space (as empty blocks) for FTL [41, 42].

Garbage Collection. When the prepared empty blocks are used up, FTL needs to *reclaim* block(s) as victims. Before reclaiming a block, if the block has valid pages, the FTL needs to reallocate such valid pages of the victims to newly-allocated blocks from the free pool, and update the mapping information before erasing the old block(s). This block reclaiming process is known as the *Garbage Collection* (GC). The page migrations (reads/writes) necessary to perform this remapping of the valid data make GC a time-consuming activity [11, 23, 12, 42, 32]. If the GC is executed too early, reclaiming blocks can prevent them from being utilized for new writes, which can in turn introduces unnecessary program/erase (P/E) cycles and long latency for I/O. Consequently, GCs in current SSDs are typically invoked *on-demand* and *delayed as much as possible*. This results in GCs being triggered at two points: 1) when FTL does not have any available free block [6]. Invocation at this point is called *free-block reclaiming GC* (*FR-GC*); and 2) when FTL has no free page in a logical block [41]. We refer to this type of GC as *logical-block reclaiming GC* (*LR-GC*), also known as block merge [25] [41].



(a) SLC SSD (120GB). (b) MLC SSD (256GB).

Figure 5: Performance difference between the worst-case and average latencies. Performance variation and the worst-case latency in SSDs may be higher than spinning disks.

Types of Address Mapping. Based on the address mapping granularity used, an FTL can be classified as 1) a block-mapping FTL (B-FTL) [25], 2) a pure page-mapping FTL (P-FTL) [6], or 3) a hybrid-mapping FTL (H-FTL) [41]. B-FTL manages address space using the block-level mapping information, which requires only a small amount of memory space, but, introduces many LR-GCs. In contrast, since P-FTL employs the finest-granular address mapping scheme, it can delay the GC until the erased blocks run out, which leads to only FR-GCs. However, it requires a larger amount of memory to maintain the mapping information, which makes it difficult to employ in commercial SSDs [17, 12, 41]. Finally, H-FTL combines advantages of B-FTL and P-FTL and can delay GC as much as possible.

3. Motivation

3.1. Garbage Collection Impact

Performance Variations. Large performance variations render widespread adoption of SSDs difficult in environments

where satisfying QoS demands is important [11, 23, 27]. Even though overall I/O throughput may be very good, higher worst-case latency experienced by some requests can easily make SSDs an I/O bottleneck as compared to disks. To analyze the performance variation problem on commercial SSDs, we performed a simple experiment. We selected SSDs of two different Samsung NAND flash types, and measured the difference between the average and worst-case latencies. We employed a new device for each test because SSD internal states change over time, which in turn impacts the I/O performance. In addition, to measure true GC effects, we sent requests, without file system or buffer cache intervention, by using Intel Iometer [18], and did not introduce any synthetic idle times to the SSDs. Figure 5 shows the performance variation (difference between the worst-case and average latencies) under various request sizes over time. Due to LR-GCs, the variations are shown from the first experiment on the device. As the execution progresses, free blocks are eventually exhausted, and the variations dramatically increase. This phenomenon is more pronounced with the Multi Level Cell (MLC) [33] based SSD since its write latency is 3.5 – 6.4 times higher than that of the Single Level Cell (SLC) [34], due to its larger page/block sizes. Even though the SLC-based SSD provides better I/O performance in the worst case, its performance still fluctuates. Note that, regardless of the NAND flash type employed, all worst-case latencies observed are over 200 *ms* (ranging from 210 *ms* to 4670 *ms*). Considering the fact that the worst-case latency of the disk in Figure 1a is around 63.2 *ms*, we see that the performance variation problem of SSDs can easily make their performance worse than that of a disk.

Write Cliff. We now demonstrate how the worst-case latencies affect the performance of state-of-the-art SSDs. For this, we selected Samsung 830 SSD (128GB) and OCZ Vertex 3 (256GB). These two SSDs have been manufactured in 2011, and employ a state-of-the-art high speed interface (SATA 6Gbps). Since the original Intel Iometer cannot capture the time series performance, we modified it in order to evaluate SSDs in every second, and applied an Iometer random write access pattern to the two SSDs. In addition, the modified Iometer periodically injected idle times – 20 seconds per 2 minutes. Figure 6 plots the worst-case latency and throughput of these two different SSDs. Even though the actual latency values with different SSDs are different from each other, their performance behaviors with respect to the worst-case latency exhibit similar patterns: *once the GC starts to reclaim blocks, the worst-case latency gets worse by about 40x, and at the same time, throughput degrades by 64%*. This severe performance drop is referred to as *write cliff*. Note that the worst-case latency values keep growing which in turn significantly reduces throughput irrespective of the idle times introduced (no background task is observed in these SSDs).

3.2. QoS-awareness and Deadlines in SSDs

Quality of Service. Due to the worst-case latency and write cliff imposed by GCs, SSDs can violate QoS in terms of the level of the guaranteed service quality. This QoS violation is a more problematic challenge in cases where multiple SSDs are used (e.g., in systems like data centers). Let p_{gc} denote

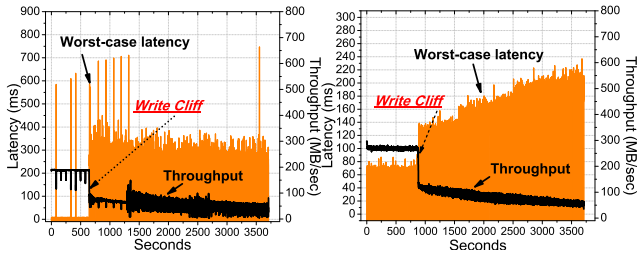


Figure 6: Write cliff of modern SSDs with a state-of-the-art high speed interface (SATA 6Gbps).

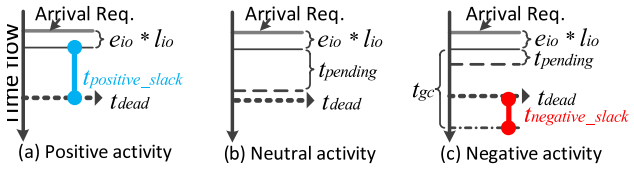


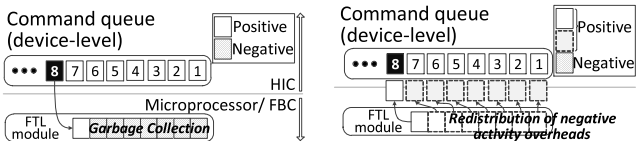
Figure 7: Illustration of different types of activities. HIOS classifies activities based on the slack time.

the probability that a GC occurs in an SSD and n indicate the number of I/O nodes in a cluster (or SSD array). Then, the probability with which a system that parallelizes data accesses among n SSDs suffers from GCs is $1-(1-p_{gc})^n$, which would be unacceptable to many data-intensive applications. Considering the Carver cluster at National Energy Research Scientific Computing Center [2] as an example, n is 16 and the p_{gc} of an SSD tested in the previous section is 0.14. In the case where Global Parallel File System (GPFS) stripes an I/O request over Carver I/O nodes, *the probability with which that request suffers from the worst-case latency on the write cliff is 91%*. It is therefore clear that satisfying QoS and providing deterministic latency by reducing the SSD performance variation are strongly desired.

Deadline Assignment. To specify QoS, users (e.g., a manufacturer, operating systems) can introduce a *maximum allowable latency* for I/O requests, with respect to the time at which an actual request is received. This deadline is typically specified at the *configuration time* through the standard thin interface protocols. However, if desired, the *deadline can be dynamically reconfigured* by leveraging existing legacy I/O command or non-data commands of such thin interface protocols [19, 36]. Consider as an example SATA 3.1 [35]. The SATA protocol already specifies the “PRIO” and “ICC” fields, which are used for deadline and priority configurations. When a host wants to configure through an I/O request, it can place ’01b’ into the PRIO field, which means "Isochronous - deadline dependent priority" and fill the corresponding deadline values in the ICC filed (8 bits). Note that the smallest unit of deadline that the current SATA provides is 10 *msec*.

4. The Proposed I/O Scheduling Strategies

The main idea behind the design of HIOS is to *classify* the I/O requests, based on their current level of deadline satisfaction, as *positive*, *negative*, or *neutral*, and schedule the sub-requests



(a) HIL with a conventional scheduling method.

(b) Redistribution of negative activity overheads.

Figure 8: Comparison between HIL with a conventional scheduler and our proposed HIOS.

associated with them (called activities)[2], such that the overheads of negative (critical) activities are *redistributed* across the positive (non-critical) activities to improve performance.

4.1. Activity Classification

To classify the impact of different activities, it is important to analyze the *slack* available to a scheduler to schedule I/O requests within a deadline. Specifically, slack-time is the duration between the end of the time to execute an I/O request and its assigned deadline. Let us assume that t_{dead} denotes the deadline, t_{gc} is the time taken by the GC, $t_{pending}$ is the time spent by the request waiting for a conflicting request to complete, e_{io} is the execution latency for a unit size I/O request, and l_{io} is the size of the I/O request. Based on these definitions, slack-time, t_{slack}, can be expressed as follows:

$$t_{slack} = t_{dead} - (t_{gc} + t_{pending} + e_{io} * l_{io}). \quad (1)$$

Considering the amount of slack-time and the status (busy/free) of the channel to which the request is heading, HIOS classifies I/O requests into three categories:

Positive Activity. Owing to the uniform and very fast access latencies of SSDs [33], I/O requests would typically have low e_{io} resulting in a positive slack, if they are not involved in t_{gc} or $t_{pending}$. We use $t_{positive-slack}$ to denote this positive slack, and the I/O requests with $t_{positive-slack}$ are classified as *positive activities* (Figure 7a). In the ideal case, we want to convert all I/O activities to positive activities.

Negative Activity. When an SSD needs to reclaim blocks to serve an I/O request, the request may have a long blocking latency due to GC. Since this blocking latency is higher than the normal I/O operation latency, the slack-time for the request has a negative value. Such a negative slack is denoted as $t_{negative-slack}$. The activities with negative slack-time that cannot be executed within their deadlines are called *negative activities*, as shown in Figure 7(c). HIOS redistributes the overheads of these negative activities in an attempt to eliminate performance variations during execution.

Neutral Activity. As shown in Figure 7b, these activities are those I/O requests that have a small positive slack ($0 < t_{slack} <$ (*# pages per block/queue size*)*e_{io}), but can easily be converted to negative activities, if any negative slack is distributed to them. Often, neutral activities have a non-zero $t_{pending}$ due to channel resource conflicts. However, the resource conflicts typically do not last more than 2 requests, rendering the overall slack given by Equation (1) as a small positive value. More importantly, if the $t_{pending}$ component of a neutral activity is not addressed, this may cascade into a negative component on other positive activities.

It should be noted that, in theory, a neutral activity can be either negative or positive based on whether or not HIOS redistributes the overheads of any negative activity to it. While HIOS handles negative activities by converting them to positive activities, it may treat neutral activities as positive activities and steal slack-time from them. However, HIOS will reorder I/O requests when there is no GC invocation in an attempt to resolve the channel resource conflict problem.

4.2. Scheduling Negative Activities

It should be emphasized that the overheads associated with negative activities are on the "critical path" leading to performance variations. As a result, they can introduce QoS violations. Let us consider Figure 8 as an example scenario where a redistribution of the overheads associated with negative activities may be of help. In Figure 8a, there are eight requests in the command queue, with the 8th request (from the head) taking the entire burden of the GC. In such a scenario, QoS violations cannot be addressed by HIL-level conventional schedulers. Regardless of the specific scheduling strategy employed, request 8 will violate its deadline (specified at the configuration time), due to the GC invocation. In contrast, in Figure 8b, HIOS splits the GC overhead associated with request 8 across seven different GC segments[3], and then redistributes them over I/O requests 1 through 7. In this way, the burden brought by the negative slack is *shared* among multiple requests having positive slacks, and we can eliminate the negative activity (request 8).

Note that simply reordering commands of the incoming I/O requests by predicting of GC would not work as it would be oblivious to which GC segments can be assigned and how many I/O requests can effectively hide these overheads, whereas our negative activity redistribution determines arrival times using the device-drive command queue and effectively amortizes its overheads over positive ones. Further, when redistributing these overheads, HIOS does *not* change the relative order of positive activities with respective to each other and does *not* require idle times for the redistribution.

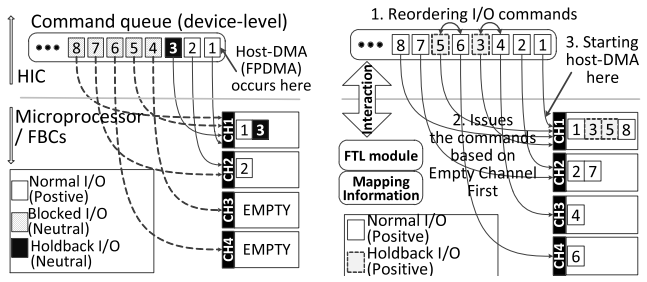
4.3. Scheduling Neutral Activities

To prevent creating "new" negative activities, HIOS carefully manages the neutral activities since they may lead to a bottleneck. Figure 9 illustrates an example scenario where a neutral activity results in performance degradation and how HIOS addresses this situation. In this example, the initial order of eight device-level commands (e.g., $1 \sim 8$) is determined by the order in which they are presented to the SSD through a thin interface, irrespective of the host-level scheduler used.

A conventional scheduling method in HIL determines an order for the I/O requests and initiates the host-DMA for *hold-back* request 3 (making other I/Os blocked), being unaware of the internal SSD architecture. Therefore, request 4 will be

[2] We use the terms "I/O request" and "activity" interchangeably.

[3] A "GC segment" is defined as the smallest unit of GC that can be paused to be resumed later.



(a) FIFO I/O scheduling (b) Channel QoS-aware scheduling

Figure 9: Illustration of performance degradation due to channel resource conflicts. While existing schedulers cannot serve recently-arrived requests 4 to 8, HIOS can immediately serve them by converting neutral activities into positive ones.

delayed in the queue on the account of the neutral activity of request 3. Similarly, requests 5∼8 would be blocked for a duration of $t_{pending}$ generated by the neutral activity. To address this problem, the best solution is to reorder I/O requests based on the channel state. Specifically, HIOS checks a flash bus controller register that maintains the state of the relevant channel. If the channel is busy serving other requests, HIOS interchanges the neutral activity with other positive activities, thereby serving the positive ones promptly. For example, in Figure 9b, requests 3 and 4 are swapped to promptly exploit the free channel 3, and then HIOS initiates the host-DMA for request 4. After that, it issues the neutral activity to the FTL. Since channel 1 would be released during the host-DMA periods of request 4, meaning that the NAND flash transactions of the issued request 1 are completed, and the neutral activity can be serviced immediately. It should be noted that, if the scheduler does not adopt this strategy, both neutral and positive activities may become negative ones. Since HIOS reorders the tag information for I/O requests by employing a device-drive command queue, computation times for reordering, host-DMA setup, and activation are not seen by the host. Note also that the host-DMA in this figure does not indicate a memory DMA or a DMA activity between NAND flash and SDRAM. Instead, it represents a method for data movement between the host and storage via a Southbridge cable, called *First-Party DMA* in NCQ (*FPDMA*). FPDMA is not a low-cost operation like memory DMA because the data require multiple (data) frames and take longer times. Therefore, HIL, ignorant of the FPDMA technology, may have to wait for the host-DMA completion, as in the case of request 4 [35, 19, 39].

5. Details of Our Scheduling Algorithm

5.1. Garbage Collection Detection

GC invocations in SSDs are generally "unpredictable" and "aperiodic" since they depend on the I/O access pattern. However, HIOS predicts GC invocations using tags associated with I/O requests *before* the host-DMA (FPDMA) engine starts a transfer. The prediction techniques employed by HIOS vary, depending on the type of GC.

Prediction. In the case of FR-GC, after checking the number of available free blocks in the reserved blocks pool of the existing FTL, HIOS determines if it is under the GC threshold. On the other hand, to predict LR-GC, HIOS first parses a tag in the command queue, and obtains the logical sector number (LSN, which is compatible with block device address) and the length of the I/O request. Once HIOS obtains the LSN and the length, it calculates the logical block number (LBN) by dividing the LSN by the number of physical blocks in the SSD's address space. The LBN related to the I/O request is translated to the corresponding physical block number(s) (PBN) by looking up the mapping table of the FTL. The number of free pages is computed by subtracting the page offset of the *pointer*, indicating the next available free page location in the block, from the total number of pages in the block. By comparing the length of the I/O request and the number of free pages, HIOS then predicts the GC demand.

Algorithm 1: Positive activity accumulation. Positive activities are accumulated before initiating the host-DMA (FPDMA) engine.

```
Input: memory layout for parsing I/O tags of pre-information
Output: the number of accumulated positive activity (n_req), list of pre-information (ncq),
        and target GC block number (gcLbn)
tag := parseTag(memory-layout)
t_gc := cacluateGCtime(tag.lsn)
t_slack := cacluateSlacktime(tag.lsn, tag.size)
if t_gc > 0 and t_slack < 0 then
    gcLbn := caculateLbn(tag.lsn)
    t_steal = t_slack /* negative slack */
    pushPreinformation(ncq, tag)
    while t_steal ≤ 0 do
        newTag := parseTag(memory-layout)
        t_slack := calcuateSlacktime(newTag.lsn, newTag.size)
        sendtoHost(tag) /* acknowledge for tag */
        if t_slack < 0 then
            /* push this tag at the end of NCQ, this is allowed
               until NCQ are not full */
            passNextAround(ncq, newTag)
        else
            push-Preinformation(ncq, newTag)
        t_steal = t_steal + t_slack
        n_req = n_req + 1
else
    ManageNeutralActivity(tag);
```

GC-Cost Estimation. After detecting a GC invocation, HIOS predicts its duration, t_{gc}, which is needed for calculating t_{slack}. Since t_{gc} is affected by the number of pages to be migrated, which is in turn determined by the number of valid pages, HIOS inspects the number of valid pages, n^{pg}_{valid}, in the logical block. Let us denote the I/O time for a page write by e^{pg}_{write}, the page read by e^{pg}_{read}, and the block erase time by e^{blk}_{erase}. Then, the time that will be taken by the GC can be expressed as: $t_{gc} = \{(e^{pg}_{write} + e^{pg}_{read}) * n^{pg}_{valid} + e^{blk}_{erase}\} * n^{blk}_{valid}$, where n^{blk}_{valid} is the number of target blocks. Whenever HIOS detects $t_{negative-slack}$ based on Equation (1), the corresponding I/O activity is classified as a negative activity.

Admission Control. The minimum execution time of a GC segment is e_{pair}, which is typically $e^{pg}_{write} + e^{pg}_{read}$, where e^{pg}_{write} and e^{pg}_{read} are the latencies for writing and reading a physical flash page, respectively. If a host sets the deadline value less than $e_{pair} + e^{blk}_{erase}$, HIOS cannot meet the deadline requirements and needs an admission control process. There are two options to here: 1) *abort NCQ subcommand* and 2) *deadline handling subcommand*. The abort NCQ subcommand discards

Algorithm 2: Negative activity redistribution. Note that GCs are executed only by the redistribution process.

Input: the number of accumulated positive activity (n_{req}), list of pre-information (ncq), target GC block number (gcLbn)

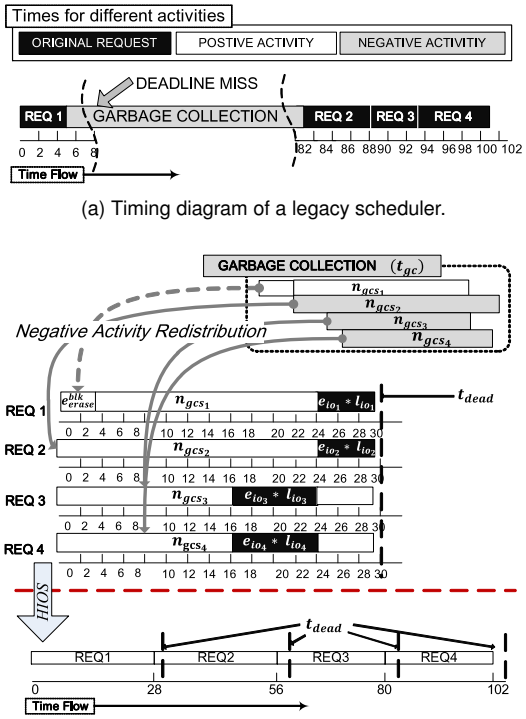
```
pbn := popBlock(freeBlockPool)
ftl.eraseBlock(pbn)
for i < n_req do
    tag := fetchPreInformation(ncq)
    t_positiveslack := calcuateSlacktime(tag.lsn, tag.size)
    n_gcs_i := t_positiveslack / e_pair
    initiateFPDMA(tag)
    /*FPDMA time and migration time can be overlapped*/
    ftl.pageMigration(pbn, gcLbn, n_gcs_i)
    if i ≠ n_req − 1 then
        ftl.pauseMigration()
    check&waitFPDMA()
    ftl.commit(tag.iotype, tag.lsn, tag.length)
    i = i + 1
pushBlock.(freeBlockPool, gcLbn)
```

all the commands in the device-level queue and stops I/O services. In comparison, the deadline handling subcommand is a best-effort service, meaning that latencies of I/O requests may be not bounded by the deadline, but we continue to service I/O requests with a failure notification. We use the latter in our evaluations, and the deadline satisfaction results are given in Section 6.3.

5.2. Negative Activity Redistribution

Slack-Stealing. After determining $t_{negative-slack}$, HIOS next determines how many non-critical I/O requests are needed to convert negative activities to positive ones. Since HIOS needs information about the I/O requests to calculate $t_{positive-slack}$, it first obtains a tag through the frame information structure (FIS), which is a data frame communicated between the host and the storage device. The tag delivers pre-information including the LSN and the size of the I/O request, through the register-FIS to the host. Algorithms 1 and 2 give the procedure employed for accumulating positive activities and the procedure employed for redistributing the overheads associated with negative activities, respectively. Once the tag is extracted by parsing the FIS, HIOS calculates $t_{positive-slack}$ for the corresponding I/O requests based on Equation (1) before the actual data transfer starts (i.e., sending DMA-setup-FIS to host). The amount of slack-time that HIOS needs to steal depends on the amount of overheads associated with the negative activities. Assuming t_{steal} denotes the amount of time to steal and n_{req} denotes the total number of I/O requests that HIOS requires for redistributing the overheads, t_{steal} can be calculated as: $t_{steal} = \sum_{i=0}^{n_{req}} t_{positive-slack} - e_{erase}^{blk}$.

Redistribution. Once HIOS determines t_{steal}, the scheduler accumulates tag commands, classified by positive activities, until we have $t_{steal} \leq t_{negative-slack}$ (of the detected negative activity). HIOS then sends back an acknowledgement for receiving the tag through the register-FIS to the host, but does not activate FPDMA for the actual data transfer. If there is a command containing LSN related to a successive GC during this accumulation process, which means another GC is invoked during the current GC, it is passed to the next accumulation round – in our observation, it can be handled by 32 NCQ entries (see Section 6.3). After HIOS collects



(a) Timing diagram of a legacy scheduler.

(b) Timing diagram with redistributing negative activities.

Figure 10: Timing diagram for 4 I/O requests. While a conventional I/O scheduler introduces a long latency for request 1 (a), by redistributing the overheads associated with negative activities, all 4 requests satisfy their deadline under HIOS (b).

tags for n_{req} number of requests in the command queue, it determines how to assign GC segments to positive activities so that they can be removed from the critical path. As indicated in Algorithm 2, for each i^{th} tag command in the queue, HIOS assigns n_{gcs_i} number of GC segments that can be executed in $t_{positivie-slack_i}$ so that the I/O time of each request can be bounded by a deadline $n_{gcs_i} = t_{positive-slack_i}/e_{pair}$, if $i \neq n_{req}$, and by a deadline $(t_{positive-slack_i} - e_{erase}^{blk})/e_{pair}$, otherwise. HIOS executes GC segments as specified by n_{gcs_i} and pauses the GC at this point. While pausing the GC, HIOS issues the non-critical I/O commands to the FTL, and then resumes the GC again. This process is repeated n_{req} times. In this redistribution process, all negative activities are transformed to positive ones by redistributing their overheads to positive activities, and the assignment overheads are hidden behind the time for handling FISs (before setting up the FPDMA).

Figure 10 illustrates how this redistribution works and how the timing diagram changes after the slack-time stealing and overhead redistribution have been carried out. While a conventional I/O scheduler would have a QoS violation when it serves request 2 due to being unaware of GC, HIOS redistributes the overheads of negative activities by stealing $t_{positive-slack}$ of accumulated tag commands. As a result, all I/O requests including request 2 are served within their corresponding deadlines. Note that, since the breakdown of GC enables HIOS to determine erase time for the reserved block and to allocate block erase in a request before starting page migration (as shown in Figure 10, request 1), it can serve

Algorithm 3: Neutral activity management. Note that I/O requests having a channel conflict will stay in NCQ for a while, thereby being served without the channel conflict.

```
PPN : = getCursorPpn(tag.lsn)
chOff := tag.lsn % # of channel /* stripping/interleaving */
chipOff := PPN % nums of chips per channel /* pipelining */
bChBusy := checkPreviousIO(chOff, chipOff)
/* neutral activity detection */
if bChBusy = true then
    if HasSamePPN(ncq, PPN) then
        aggregateCommand(ncq, tag)
    else
        /* Reordering this tag without DMA starts */
        pushPreinformation(ncq, tag)
        tag := getNextPositiveActivty()
        if tag = null then
            tag := popVictim(ncq);
sendtoHost(tag)
initiateFPDMA(tag)
ftl.commit(tag.iotype, tag.lsn, tag.length)
```

negative requests and execute page migrations by interleaving the GC segments with positive ones. In the worst case, HIOS starts redistributing GC when FTL has n_{valid}^{blk} blocks in a free block pool, so that block erasing and valid page migration can be interleaved among negative and positive ones in a critical path. Therefore, GC overheads can be distributed among I/O requests in the command queue.

5.3. Neutral Activity Reordering

Since neutral activities are comprised of potential blocking I/Os due to the channel resource conflicts, HIOS interchanges the neutral activities with positive ones at runtime. We want to emphasize that this process is invoked *only if* HIOS does not have to deal with additional overheads of GC at that time. To prevent the creation of blocking I/Os, HIOS inspects the channel states before reordering the commands.

Channel Conflict Detection. To detect a channel resource conflict, HIOS first secures the channel offset and physical page number (PPN) for the next available free page. We assume that the FTL employs channel striping and interleaving techniques (see Section 2.2) at a superpage granularity, so that I/Os can be scattered across all channels. Therefore, channel offset for each scattered data can be determined by LSN modulo the number of channels. We further assume that channel pipelining is employed within a single channel to fully exploit internal parallelism. In this method, PPN indicates the flash chip location in a single channel by chip offset, which is obtained as PPN modulo the number of chips. To combine the channel and chip offsets, HIOS speculates which channel and flash chip will be used for serving the corresponding requests. HIOS then inspects the flash bus controller register, which contains the states associated with the channel offset and maintains the state information of the channel. Since this detection process can be executed before determining the actual data arrival, its overheads can be hidden to alleviate performance variations coming from the channel contentions.

Contention Alleviation. After inspecting the channel states, HIOS arbitrates channel contention by employing two techniques: 1) *I/O reordering based on an empty-channel-first* (*ECF*) policy, and 2) *page-based aggregation*. Algorithm 3 formalizes these two techniques. First, HIOS checks if the incoming I/O requests would result in any channel conflicts. If so, it adds the request into the command queue, and then gets the next positive activity from the queue. To hide the overhead of retrieving the next positive activity, HIOS sends an acknowledgment via the FIS. If there is no I/O request waiting to be serviced and no positive one, HIOS issues the enqueued neutral activity. In this case, HIOS cannot invoke the ECF policy. Otherwise, HIOS issues the retrieved positive activity destined for an empty channel to existing FTL. While the neutral activity is residing in the queue, the previous I/O request for the channel would be completed (i.e., the channel will be free). Consequently, that neutral activity will transform into a positive one at runtime. In addition, while HIOS schedules a command, if it receives another command containing an adjacent address within a superpage boundary heading toward the same channel with the neutral one, it aggregates these two commands into a single request. This process, called *page-based aggregation*, helps the scheduler reduce the number of I/O requests going to a busy channel.

6. Experimental Evaluation

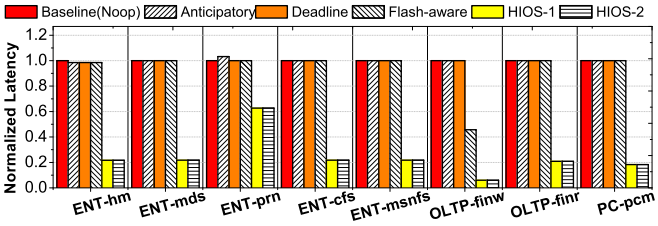
Methodology. Garbage collection (GC) invocations and performance could significantly vary based on the underlying physical data layout. The ideal experimental methodology in evaluating GCs in a fair fashion would be to prepare different sets of pristine devices for every single evaluation step with different software stacks and system parameters such as different I/O schedulers, FTLs, deadline requirements and GC thresholds. Unfortunately, *there exists no commercial SSD that allows us to modify its internal software stack and system parameters*. Moreover, preparing a pristine device for every evaluation step is not practical. Therefore, we performed simulation based study with real applications workloads extracted from different application domains.

Simulator. We performed an event-driven simulation, which models bus transaction-level flash chips as well as data paths and fully implements a typical SSD storage software stack, including a flash interface logic, host interface logic, flash translation layers, and corresponding garbage collectors. The entire code for flash software in the stack is around one hundred thousand of lines. The simulated SSD, which represents state-of-the-art, consists of 8 channels, each of which is connected to two different NAND flash devices (MLC [33]/SLC[34]); unless otherwise stated, MLC is used as default.

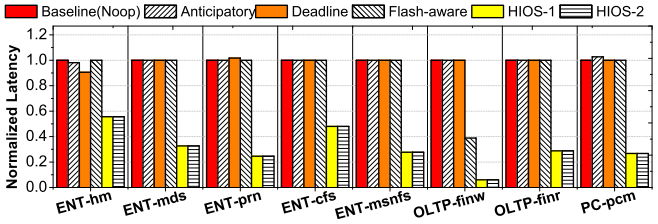
HIL-based I/O Schedulers. We implemented three state-of-the-art I/O schedulers (based on noop, anticipatory, deadline I/O scheduling algorithms [29]) and a flash-aware I/O scheduler [24] in HIL to perform a quantitative comparison. *Noop* simply submits I/O requests to the underlying FTLs on a FIFO basis. *Deadline* sorts the request in the device-level queue based on LBA (ascending order) and commits them based on the order in the queue with preference of read requests (preventing the starvation of the requests). *Anticipatory* also exploits the sorted queue, but waits for future requests that target spatially closer data. In contrast, the *flash-aware scheduler* [24] attempts to reduce GC overheads by bundling the write requests based on the block boundary adopted by the

Workload			Baseline Latency (Noop)	
Type	Name	% Writes	Avg. (μs)	Worst-case (μs)
Enterprise (ENT)	Hardware Monitor (hm)	6.14	1,194	149,454
	Media Server (mds)	1.74	408	145,636
	Printer Server (prn)	14.51	684	280,297
	MSN Media Server(cfs)	37.51	758	145,636
	MSN File Server(msnfs)	50.04	647	145,626
OLTP	Financial Write (finw)	84.60	1,868	524,716
	Financial Read (finr)	21.54	1,685	151,669
Desktop(PC)	PCMARK05(pcm)	53.07	569	175,791
HPC	HPIO on 2 cores (hpio2)	100	1,772	159,709
	HPIO on 4 cores (hpio4)	100	1,785	303,335
	HPIO on 8 cores (hpio8)	100	1,814	292,271
	IOR	100	1,814	303,335

Table 1: Characteristics of our workloads. The third column gives the percentage of writes in each application.



(a) B-FTL



(b) H-FTL

Figure 11: Worst-case latencies for two different types of FTLs (normalized to the noop scheduler whose absolute values are given in Table 1).

underlying FTL. Note that all these I/O schedulers employ standard 32 NCQ entries [19] and enjoy internal parallelism by exploiting interleaving, striping, and pipelining (see Figure 4). HIOS uses 30 *ms* as the *default deadline* (target QoS value); we later perform a sensitivity study with other deadline values. To better understand the latency impacts of different activities, we tested two different implementations of HIOS: 1) **HIOS-1** manages only negative activities, and 2) **HIOS-2** manages *both* negative and neutral activities (unless otherwise stated, HIOS indicates HIOS-2).

Flash Translation Layers. All I/O schedulers tested in this work are implemented on top of two FTLs: Block-mapping FTL (*B-FTL*) [25] and Hybrid mapping FTL (*H-FTL*) [41]. Each FTL has 6% reserved free blocks of the total storage area, and the default FR-GC threshold is set to 1%. We also perform a sensitivity study with other threshold values.

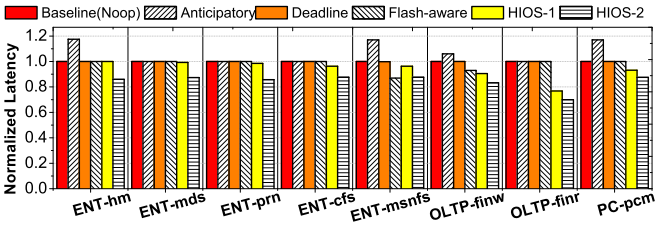
Workloads. To evaluate HIOS on a diverse set of I/O patterns, we used workloads coming from different application domains. The first one includes realistic enterprise workloads [31] published by SNIA IOTTA [37]. In addition, two commercial OLTP I/O traces [4] are used. We also evaluated HIOS when running two high performance computing (HPC) I/O workloads: HPIO [40] and IOR [5]. For these applications, we collected the block-level I/O information using *blktrace* [7]. We also used the pcmark05 benchmark [3], as a representative of the desktop (PC) workloads. This test suite includes application loading, general usage, file writes, operating system start-up and virus scan workloads. Finally, to measure the time-varying performance degradation and the effects of queue size variations for neutral activities, we used *Intel Iometer* [18]. This tool helps us test sequential and random I/O patterns without any host file system intervention. To collect statistics for pcmark05 and Iometer at runtime, we employed *DiskMon* [1]. The important characteristics of our workloads are given in Table 1.

6.1. Latency Analysis

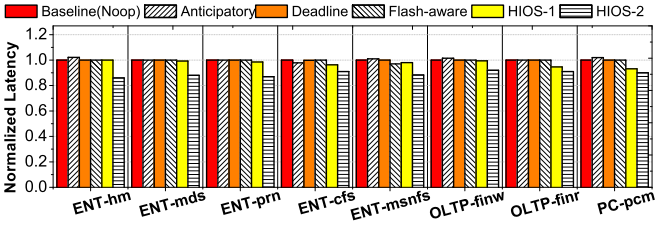
Worst-Case Latency (WCL). Figure 11 plots the *normalized* WCLs for all I/O schedulers tested with two different FTLs for all our workloads except HPC ones, which we will discuss separately. The WCLs of existing I/O scheduling polices are all similar as they are optimized mainly for random accesses. Since these I/O schedulers are oblivious to GC, they cannot meet a specified deadline if negative activities take place during execution. For a majority of our workloads, changes in the I/O access patterns or temporal/spatial locality do not impact the WCL, and we can see that the WCL is slightly worse than or equal to that of the noop scheduler. In contrast, the flash-aware I/O scheduler provides about 58% improvement in the WCL in the case of finw. Since finw has about 84% write operations, bundling write requests considering block boundaries maximizes page utilization for a logical block and helps to prevent the LR-GC invocations, compared to other schedulers. However, the flash-aware scheduler does not perform well with other workloads since they do not exhibit block-level locality. We also see that *HIOS reduces the WCL significantly for all I/O workloads*. Specifically, we see WCL improvements ranging from 41.1% to 92.3%. Since the WCLs are handled by redistributing negative activities rather than reordering neutral ones, HIOS-1 and HIOS-2 behave very similarly (Figure 11).

Average Latency. Figure 12 shows the *normalized* average latencies. As can be observed, the improvements in average latencies brought by HIOS-1 are not significantly higher than those obtained when using the other schedulers. This is because HIOS-1 redistributes the overheads associated with negative activities from one part of execution to another, and this does not affect the average latencies much. In addition to mitigating the GC overheads, HIOS-2 improves the average latency by about 13.2% compared to HIOS-1, indicating that ECF helps HIOS achieve better performance on a workload that does not have GCs by resolving resource conflicts. Note that, while the flash-aware scheduler improves the performance of only msnfs under B-FTL, HIOS-2 outperforms all other schedulers tested on all the workloads.

HPC workloads. These workloads are optimized for parallel file systems and high-level I/O libraries such as MPI-IO [40]. Even though HPIO and IOR mostly perform accesses to non-contiguous sections of the memory space, MPI-IO changes



(a) B-FTL



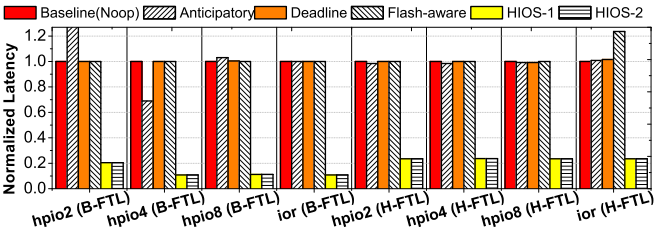
(b) H-FTL

Figure 12: Average latencies for two different types of FTLs (normalized to the noop scheduler).

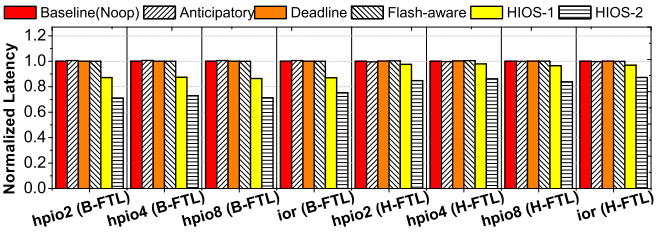
these accesses to sequential ones. Nevertheless, since HPIO has an I/O pattern with lots of redundant in-place updates and IOR has an access pattern dominated by a large number of accesses to a small address range, the WCLs observed with these HPC applications are similar to those observed with the enterprise applications. Figure 13 illustrates the worst-case and average latencies for the HPC workloads. In Figure 13a, the reason for high WCLs with conventional I/O schedulers is the fact that these schedulers try to reshape access patterns using spatial/temporal locality even when accesses exhibit high sequentiality. Compared to the conventional schedulers, HIOS offers a 86.6% shorter WCL on average by stealing positive slack-times from select I/O requests and redistributing the overheads associated with negative activities. One can also observe that the average latencies with HIOS-2 are about 30% better than the other schedulers tested as a result of the page-based aggregation, which has an impact on de-duplicating the I/O contents. Overall, these results clearly demonstrate the effectiveness of HIOS in reducing the WCLs.

6.2. Performance Variation Analysis

Figure 14a plots the standard deviation of access latencies with all the workloads tested under B-FTL. As stated earlier, the conventional I/O schedulers are not aware of the flash and GC characteristics, and this results in very high variations in performance. Even though the flash-aware scheduler [24] tries to reduce the GC overheads by bundling the write requests, it cannot alleviate the performance variations when some of the GC activities interfere with critical requests, turning them into negative activities. This is because the flash-aware scheduler does *not* explicitly address the GC interference issue and the available queue size is limited to take any advantage of the block boundary-based locality to reduce the number of GC invocations. Moreover, the average latency with the flash-aware I/O scheduler is not very different from the average latency achieved by the other legacy schedulers tested. In contrast to the WCL in the case of the HPC workloads, in most



(a) Worst-case latency



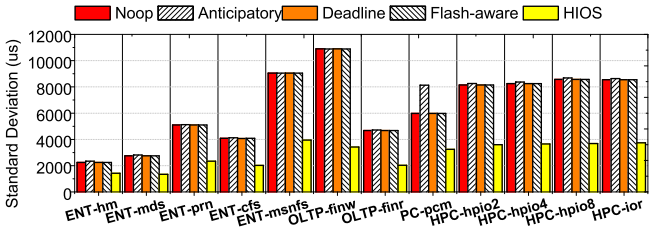
(b) Average latency

Figure 13: Latency comparison under the HPC workloads.

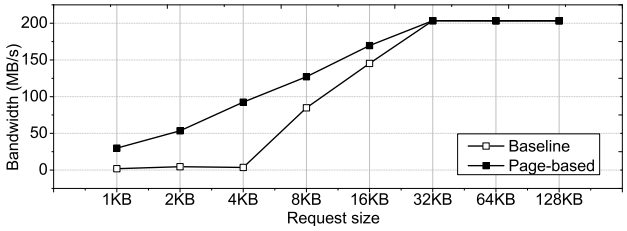
conventional I/O schedulers, the standard deviation does not vary much across different schedulers (even though it is high in absolute terms). This is because the access pattern in HPC applications is already optimized by MPI-IO and any further reshaping of this pattern by a conventional I/O scheduler has little impact on reducing the GC overheads. As a consequence, the standard deviation is quite stable across all legacy I/O schedulers. However, since HIOS converts negative activities to positive ones, the standard deviation observed with it is lower than those observed with the legacy I/O schedulers. On an average, HIOS reduces the standard deviation of access latencies by 53.1%, 58.2%, 53.9% and 51.4%, as compared to the noop, anticipatory, deadline, and flash-aware schedulers, respectively. Figure 14b gives a comparison of the baseline HIOS and the page-based aggregation version. Since the aggregation reduces the number of page writes, the throughput improves for sequential I/O requests that fall between 1 KB and 16 KB.

6.3. Dynamics and Deadline Satisfaction

To evaluate the dynamics of HIOS and deadline satisfaction based on different storage space utilization, we used an HPC workload. To have a long running time for testing the effectiveness of HIOS with multiple GC invocations, we generated a write-intensive workload using Iometer [18]. This synthesized I/O pattern consists of 80% random writes, and the average request size is 500KB. Figure 15a plots the space-varying performance for this workload. It can be observed that, as the execution progresses, the WCLs under the legacy schedulers fluctuate more. HIOS, however, uniformly distributes the overheads associated with negative activities and does not suffer heavy penalties as far as the WCLs are concerned. Figure 15b shows the percentage of total I/O requests that *miss* their deadlines under different I/O schedulers in HIL. When the SSD is in its pristine state (the space utilization is under 40% of total storage space), the percentage of the deadlines missed is just below 4% under all I/O schedulers. However, as available space runs out, the percentage of deadline misses

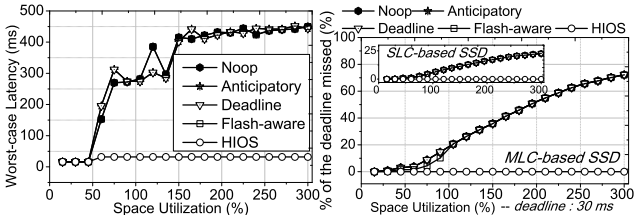


(a) Standard deviation of access latency



(b) Impact of page-based aggregation

Figure 14: Performance variation analysis of the enterprise and HPC workloads.

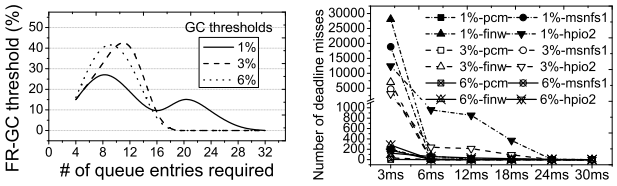


(a) Space-varying performance. (b) % of total I/O requests that miss the deadline.

Figure 15: Space-varying performance and deadline satisfaction analysis. While existing I/O schedulers with NCQ cannot serve I/O requests with a deadline of 30ms, HIOS meets all deadlines, irrespective of the NAND type employed.

dramatically increases (up to 72.3%). This deadline misses are also observed in the SLC-based SSD (the percentage of deadline misses increase up to 22.5%). This is because the number of available free pages and blocks gets smaller over time, leading to more frequent GCs (both FR- and LR-GCs). Irrespective of the type of the NAND flash employed, HIOS successfully satisfies the deadline requirements.

To measure the sensitivity of HIOS to the deadline value (QoS value) chosen, we quantified the number of deadline misses experienced with each scheduler by varying the deadline value and the FR-GC threshold. Figure 16b plots the number of missed deadlines under the different deadline values and FR-GC thresholds when using HIOS. One can observe from these results that HIOS does not miss many deadlines, until the deadline value gets very small (less than 30 ms), for all the GC threshold values (1%, 3%, 6%). Further, as can be observed from Figure 15b, while the conventional schedulers in HIL could not satisfy a 30ms deadline, no deadline violation occurs when using HIOS. Figure 16a plots sensitivity to the number of queue entries required for redistributing the GC overheads based on varying FR-GC thresholds. As



(a) Sensitivity to the number of queue entries and GC threshold.

(b) Sensitivity to deadline period and GC threshold.

Figure 16: Sensitivity to the number of queue entries, deadline period, and GC threshold.

we decrease the FR-GC threshold (6%∼1%), the number of queue entries required increases due to successive GC invocations. However, HIOS successfully redistributes them using 32 entries.

7. Related Work and Discussion

Scheduler Baseline Location. One of the reasons why one would prefer HIOS in HIL instead of the underlying flash modules (e.g., FTL, FIL), is that the deadline assignment and parsing of I/O requests need to interact with physical (PHY) and link layers of the host interface [35, 38]. While the PHY and link layers are handled by HIL, all the underlying flash modules work on the application (APP) layer, which is ignorant of the specific interface protocol and the device-level command queue handling policy used. To schedule the device-level queue from an underlying flash module, it is required to have extra queues and hardware/software logic. For example, even though out-of-order scheduling flash firmware (ignorant of the PHY and link layers) has no idea about slack-time stealing (using PHY-level information) and GC redistribution, it needs extra in-bound and out-bound queues to schedule I/O requests [30] or software stack redesign [22]. In contrast, a HIL-based scheduler (such as HIOS) controls the PHY and link layers and directly retrieves information from the host interface. Hence, it can schedule I/Os in a QoS- and GC-aware fashion.

TRIM Command. OS can inform the SSD which files are no longer considered in use using a TRIM command [19]. Even though the TRIM command is an effective way of alleviating the GC overheads, an SSD still requires GC to perform page migrations. This is because the pages invalidated by the TRIM commands are distributed among multiple physical blocks. In addition, the effectiveness of the TRIMs depends primarily on the file system status and user patterns. In contrast, HIOS distributes the GC overheads in a file system status independent fashion, and hides them from users. Further, in cases where blocks have many pages, which are invalidated by the TRIM commands, HIOS can quickly finish the GC-redistribution process, and it needs a shorter slack-time than the usual case because the amount of page migration is reduced by the TRIM.

Background GC (BGC). We demonstrated in Section 3.1 that on-demand GC is necessary to claim free blocks in cases where a lot of idle times are experienced. We believe that there are three reasons why BGC is difficult to employ in commercial SSDs: First, BGC can be performed only if there is a victim block that is already fully utilized. If the BGC

performs block cleaning on the underutilized blocks, it may increase *the P/E cycles* which can in turn shorten SSDs' life. Second, performing BGC incurs *extra power consumption* in reclaiming blocks during idle times. SSD makers would prefer to save power and make SSDs sleep when they are idle. Third, in cases where idle times are too short, BGC can neither immediately respond to incoming I/O requests nor be executed. In contrast, HIOS makes on-demand GCs invisible to users by redistributing them over the non-critical I/Os. This GC redistribution process does not require any idle time or extra P/E cycles in order to hide GC overheads.

Pausing and Partial GC (PGC). Pausing a GC is unsafe if there is no chance to resume that GC soon. To address this challenge, [12] requires 15%$\sim$ 20% extra flash buffer blocks (used for I/O service while a GC is suspending) in addition to the empty blocks, reserved by the existing FTL. This extra flash block requirement will be unacceptable in many cases due to cost concerns. [26] allows SSDs to serve only read requests during the GC-pause periods so that the extra flash buffer block requirement can be dropped. [27] leverages free blocks in the FTL. However, to maintain the mapping-consistency and data-integrity, it is only applicable when we use a pure page mapping algorithm, whose memory requirements are also unacceptable to many SSD appliances [17]. Further, all these PGC based efforts are not aware of QoS (deadline) requirements and do not consider the resource contention problem. In contrast, HIOS detects the GC overheads, and redistributes them over the non-critical requests, under any type of FTL, based on the defined deadline. HIOS guarantees to resume GC whenever the positive activities are served in GC-retribution process. In addition, it schedules the device-level commands by being aware of the internal resource configuration, thus making the latencies of the SSD more predictable and improving I/O performance.

8. Concluding Remarks

We proposed a novel host interface I/O scheduler (HIOS) for SSDs, which can distribute GC overheads across non-critical requests and address write resource contention problems during writes. Our experiments reveal that the worst-case I/O latencies do not exceed specified deadlines when HIOS is employed. Further, with diverse system configurations such as different flash types, FTL types, GC thresholds, and space utilizations, HIOS reduces the standard deviation of latency by 52.5%, and the worst-case latency by 86.6%, over existing I/O schedulers used in commercial systems.

Acknowledgements

This research is supported in part by NSF grants 1213052, 1302557, 1017882, 0937949, 0833126, and University of Texas at Dallas Start-up Grants.

References

[1] "http://download.sysinternals.com/files/diskmon.zip."
[2] "http://www.nersc.gov/users/computational-systems/carver/."
[3] "http://www.futuremark.com/products/pcmark05/," 2005.
[4] "http://traces.cs.umass.edu/index.php/storage," 2007.
[5] "IOR user guide." LLNL, 2007.
[6] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *USENIX ATC*, 2008.
[7] A. Brunelle, *Block I/O Layer Tracing*, 2006.
[8] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing," in *Proceedings of SC*, 2010.
[9] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *MICRO*, 2010.
[10] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications," in *ASPLOS*, 2009.
[11] L.-P. Chang and T.-W. Kuo, "Real-time garbage collection for flash-memory storage systems of real-time systems," *TECS*, 2004.
[12] S. Choudhuri and T. Givargis, "Deterministic service guarantees for NAND flash using partial block cleaning," in *CODES+ISSS*, 2008.
[13] P. Desnoyers, "Empirical evaluation of nand flash memory performance," in *HotStorage*, 2009.
[14] ——, "Analytic modeling of ssd write performance," in *SYSTOR*, 2012.
[15] C. Dirik and B. Jacob, "The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization," in *ISCA*, 2009.
[16] L. M. Grupp, J. D. Davis, and S. Swanson, "The harey tortoise: Managing heterogeneous write performance in ssds," in *USENIX ATC*, 2013.
[17] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *ASPLOS*, 2009.
[18] Intel, *Iometer User's Guide*, 2003.
[19] Intel and Seagate, *Serial ATA NCQ*.
[20] M. Jung, S.-J. Jang, S.-C. Kim, and C. ik Park, "Memory system and data storing method thereof," *U.S. Patent*, vol. 20090248987.
[21] M. Jung and M. Kandemir, "Revisiting widely held ssd expectations and rethinking system-level implications," in *SIGMETRICS*, 2013.
[22] M. Jung, E. H. Wilson, III, and M. Kandemir, "Physically addressed queueing (paq): Improving parallelism in solid state disks," in *ISCA*, 2012.
[23] M. Jung and J. Yoo, "Scheduling GC opportunistically to reduce worst-case I/O performance in solid state disks," in *IWSSPS*, 2009.
[24] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drives," in *EMSOFT*, 2009.
[25] J. Kim, J. M. Kim, S. Noh, S.-L. Min, and Y. Cho, "A space-efficient flash translation layer for Compact Flash systems," in *TOC*, 2002.
[26] Y.-G. Kim, K.-A. Kim, J. hyuk Kim, and C. ik Park, *Incremental Merge Methods and Memory Systems Using the Same*. U.S. Patent #2006004971A1, 2006.
[27] J. Lee, Y. Kim, G. Shipman, S. Oral, F. Wang, and J. Kim, "A semi-preemptive garbage collector for solid state drives," in *ISPASS*, 2011.
[28] S. Lee, K. Fleming, J. Park, , K. Ha, , A. Caulfield, S. Swanson, Arvind, and J. Kim, "BlueSSD: An open platform for cross-layer experiments for NAND flash-based SSDs," in *WARP*, 2010.
[29] R. Love, "I/O schedulers," *Linux Journal*, 2004.
[30] E. H. Nam, B. Kim, H. Eom, and S.-L. Min, "Ozone (o3): An out-of-order flash memory controller architecture," *TOC*, 2011.
[31] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to SSDs: Analysis of tradeoffs," in *EuroSys*, 2009.
[32] J. T. Robinson, "Analysis of steady-state segment storage utilizations in a log-structured file system with least-utilized segment cleaning," *SIGOPS*, 1996.
[33] Samsung, "K9GAG0B0M http://samsung.com," 2008.
[34] ——, "K9K8G08U1M http://samsung.com," 2008.
[35] SATA-IO, *Serial ATA Revision 3.1*, 2011.
[36] Seagate, *SCSI Commands Reference Manual*, 2006.
[37] SNIA, "http://iotta.snia.org/." IOTTA repository, 2006.
[38] T10, "SCSI storage interfaces," 2009.
[39] T13, *AT Attachment Model*, 2006.
[40] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *MPP*, 1999.
[41] J. uk Kang, H. Jo, J. soo Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash," in *EMSOFT*, 2006.
[42] X. yu Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *SYSTOR*, 2009.